1. 上课约定须知

课程主题：Flink源码解析 -- 第五次课（StreamTask 启动，初始化，执行）
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次内容总结

今天的课程是 Flink 源码剖析 的第五次：主要讲解 Flink 最核心的功能：StreamTask 的初始化和执行，在这之前，把 Flink 的 集群启动 和 Slot管理 还有 Job 的提交执行都做了分析。

```
1、Flink 集群服务端处理 JobSubmit
2、Slot 管理（申请）源码解析
3、部署 Task 提交到 TaskExecutor 执行源码剖析
```

本次课，就从 StreamTask 的具体执行来聊起，看看 StreamTask 是怎么初始化，怎么启动，怎么进行数据交换的。

3. 本次内容大纲

今天主要讲解的是 StreamTask 的执行：

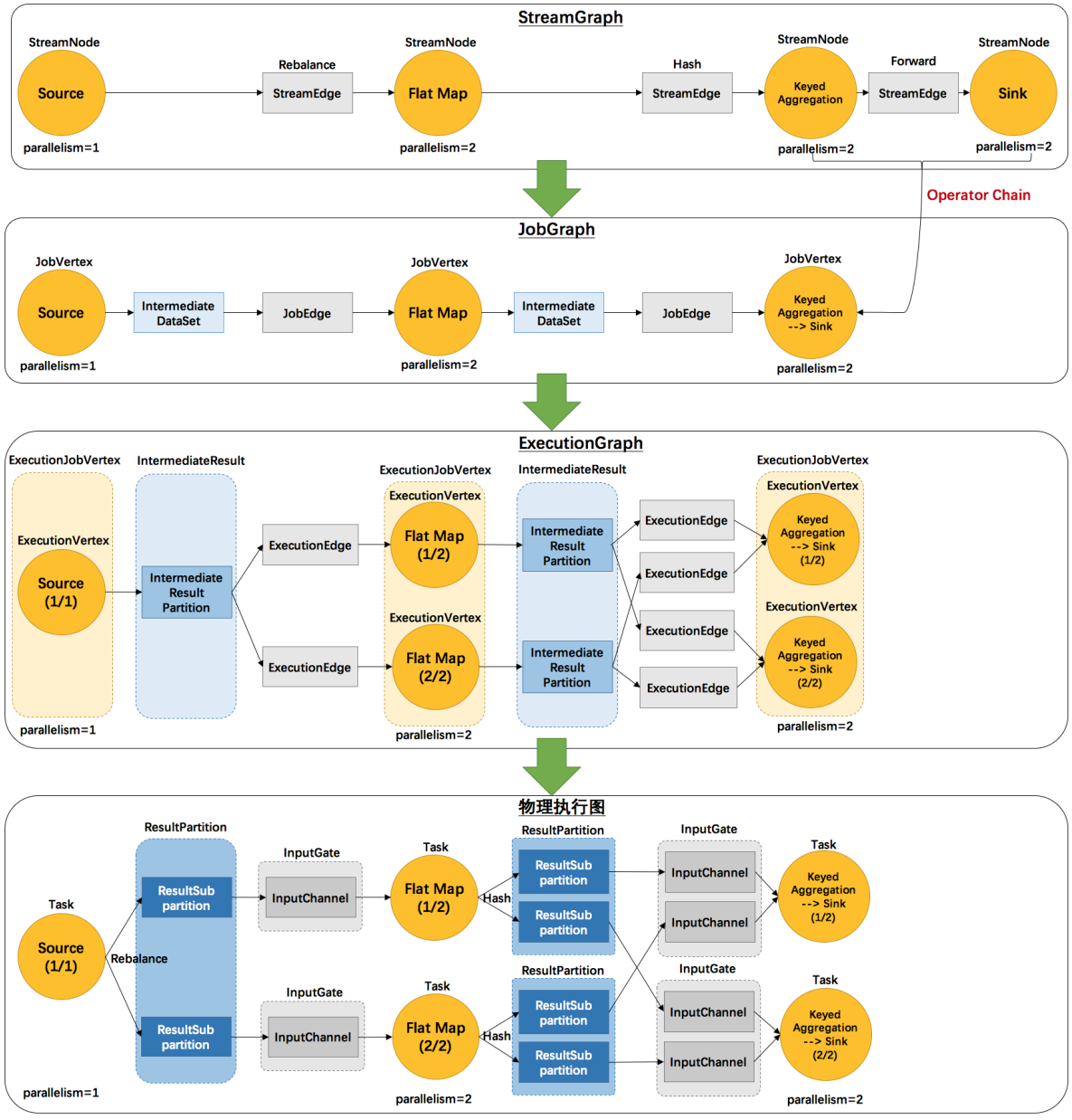
```
1、StreamTask 启动
2、StreamTask 初始化
3、StreamTask 执行
```

整体来说，在讲完 Flink standalone 集群启动讲完之后，讲解了 Flink 的应用程序 job 是如何被构建以及提交到 Flink 集群中去运行的。这次课程主要讲解的是 StreamTask 具体的运行细节。

4. 详细课堂内容

4.1. 复习一个知识

Flink 的四层图结构：



复习的最大目的：

```
1、InputGate 负责 Task 的输入
    由于可能会从多个上游Task拉取输入，所以跟每一个上游Task建立的建立的管理抽象叫做：
InputChannel
2、ResultPartition 负责 Task 的输出
    由于当前这个Task可能输出的数据要被分发到下游的多个 Task，这就证明一个 Task 有多个输出分
区，每个输出分区的管理抽象：ResultSubPartition
```

最重要的知识点，就是，由于本次课会讲解： StreamTask 的启动，初始化，执行三大核心细节！必然会涉及到： InputGate 和 ResultPartition 的初始化！

今天的额知识点： 从 从节点 接收到一个 执行 Task 的RPC 请求开始！

```
TaskExecutor.submitTask(TaskDeploymentDescriptor tdd, JobMasterId jobMasterId,
Time timeout);
```

4.2. StreamTask 部署启动

4.2.1. TaskExecutor 执行一个 Task

当 TaskExecutor 接收提交 Task 执行的请求，则调用：

```
TaskExecutor.submitTask(TaskDeploymentDescriptor tdd, JobMasterId jobMasterId,
Time timeout);
```

最重要的一些细节动作：

```
// 构造 Task 对象
Task task = new Task(jobInformation, taskInformation, ExecutionAttemptId,
AllocationId, SubtaskIndex, ....);

// 启动 Task 的执行
task.startTaskThread();
```

在该方法的内部，会封装一个 Task 对象，在 Task 的构造方法中，也做了一些相应的初始化动作，我们来看：

```
public Task(.....){
    // 封装一个 Task信息对象 TaskInfo，（TaskInfo, JobInfo，JobMasterInfo）
    this.taskInfo = new TaskInfo(....);

    // 各种成员变量赋值
    ......

    // 一个Task的执行有输入也有输出： 关于输出的抽象： ResultPartition 和
ResultSubPartition（PipelinedSubpartition）
    // 初始化 ResultPartition 和 ResultSubPartition
    final ResultPartitionWriter[] resultPartitionWriters =
shuffleEnvironment.createResultPartitionWriters(....);
    this.consumableNotifyingPartitionWriters =
ConsumableNotifyingResultPartitionWriterDecorator.decorate(....);
```

```
    // 一个Task的执行有输入也有输出： 关于输入的抽象： InputGate 和 InputChannel（从上有一个Task节点拉取数据）
    // InputChannel 可能有两种实现： Local  Remote
    // 初始化 InputGate 和 InputChannel
    final IndexedInputGate[] gates = shuffleEnvironment.createInputGates(.....);

    // 初始化一个用来执行 Task 的线程，目标对象，就是 Task 自己
    executingThread = new Thread(TASK_THREADS_GROUP, this, taskNameWithSubtask);
}
```

封装一个 Task 的时候，调用构造方法执行，会去初始化该 Task 的输入（InputGate 和 InputChannel）和输出（ResultPartition 和 ResultSubPartition）组件相关。然后初始化用来执行该 Task 的一个线程。

总之，都是通过封装一个 Task 对象，内含一个 executingThread，它的目标对象，就是 Task，所以在构造好了 Task 之后，调用：

```
task.startTaskThread();
```

之后，跳转到 Task.run() 方法，从此，真正开始一个 Task 的启动和执行。

4.2.2. Task 线程启动

Task 的启动，是通过启动 Task 对象的内部 executingThread 来执行 Task 的，具体逻辑在 run 方法中：

```
Task.run();
    Task.doRun();
        // 1、先更改 Task 的状态： CREATED ==> DEPLOYING
        transitionState(ExecutionState.CREATED, ExecutionState.DEPLOYING);

        // 2、准备 ExecutionConfig
        final ExecutionConfig executionConfig =
serializedExecutionConfig.deserializeValue(userCodeClassLoader);

        // 3、初始化输入和输出组件，拉起 ResultPartition 和 InputGate
        setupPartitionsAndGates(consumableNotifyingPartitionWriters,
inputGates);

        // 4、注册 输出
        for(ResultPartitionWriter partitionWriter :
consumableNotifyingPartitionWriters) {

 taskEventDispatcher.registerPartition(partitionWriter.getPartitionId());
        }
        // 5、初始 环境对象 RuntimeEnvironment，包装在 Task 执行过程中需要的各种组件
        Environment env = new RuntimeEnvironment(jobId, vertexId, executionId,
....);

        // 6、初始化 调用对象
        // 两种最常见的类型： SourceStreamTask、OneInputStreamTask、
TwoInputStreamTask
        // 父类： StreamTask
```

```
        // 通过反射实例化 StreamTask 实例(可能的两种情况： SourceStreamTask，
OneInputStreamTask)
        AbstractInvokable invokable =
loadAndInstantiateInvokable(userCodeClassLoader, nameOfInvokableClass, env);

        // 7、先更改 Task 的状态： DEPLOYING ==> RUNNING
        transitionState(ExecutionState.DEPLOYING, ExecutionState.RUNNING);

        // 8、真正把 Task 启动起来了
        invokable.invoke();

        // 9、StreamTask 需要正常结束，处理 buffer 中的数据
        for(ResultPartitionWriter partitionWriter :
consumableNotifyingPartitionWriters) {
            if(partitionWriter != null) {
                partitionWriter.finish();
            }
        }

        // 10、先更改 Task 的状态： RUNNING ==> FINISHED
        transitionState(ExecutionState.RUNNING, ExecutionState.FINISHED);
```

根据上述代码的执行可知：一个 Task 的状态周期：

```
CREATED ---> DEPLOYING -----> RUNNING  ----> FINISHED
```

内部通过反射来实例化 AbstractInvokable 的具体实例，最终跳转到 SourceStreamTask 的构造方法，同样，如果是非 SourceStreamTask 的话，则跳转到 OneInputStreamTask 的带 Environment 参数的构造方法。

关于一些概念：

```
1、Execution： 在 ExecutionGraph 中，其实一个 Task 的完整抽象，就是一个
ExecutionVertex，如果这个 ExecutionVertex 被尝试执行一次，就会生成一个 Execution，会被
指定一个全局唯一的 AttempID 来标识
```

4.3. StreamTask 初始化

这个地方的初始化，指的就是 SourceStreamTask 和 OneInputStreamTask 的实例对象的构建！

Task 这个类，只是一个笼统意义上的 Task，就是一个通用 Task 的抽象，不管是批处理的，还是流式处理的，不管是 源Task， 还是逻辑处理 Task， 都被抽象成 Task 来进行调度执行！

4.3.1. SourceStreamTask 和 StreamTask 初始化

启动一个 Task 的执行，这个 Task 有可能是 SourceStreamTask，也有可能是非 SourceStreamTask（比如 OneInputStreamTask，TwoInputStreamTask）等。

首先需要了解的第一个知识点：在最开始一个 job 提交到 Flink standalone 集群运行的时候，在 client 构建 StreamGraph（顶点是 StreamNode，边是 StreamEdge） 的时候，会根据用户调用的算子生成的 Transformation 为 StreamGraph 生成 StreamNode，在生成 StreamNode 的时候，会通过 OpearatorFactory 执行判断：

- 如果该 StreamOperator 是 StreamSource 的话，就会指定该 StreamTask 的 invokableClass 为 **SourceStreamTask**
- 否则为 **OneInputStreamTask**，TwoInputStreamTask， StreamTask 等。

核心代码是：

```
StreamGraph.addOperator(....){
    invokableClass = operatorFactory.isStreamSource() ? SourceStreamTask.class :
OneInputStreamTask.class;
}
```

所以当 ExecutionVertex 真正被提交到 TaskExecutor 中运行的时候，被封装的 Execution 对应的 Task 类的启动类 AbstractInvokable 就是在构建 StreamGraph 的时候指定的对应的 invokableClass。所以：

```
1、如果启动 SourceStreamTask，则启动类是：SourceStreamTask
2、如果启动非 SourceStreamTask，则启动类是：StreamTask
```

所以咱们首先来看 SourceStreamTask 的构造过程。来看 SourceStreamTask 的构造方法：

```
public SourceStreamTask(Environment env) throws Exception {
    this(env, new Object());
}
```

然后跳转到重载构造：

```
private SourceStreamTask(Environment env, Object lock) throws Exception {

    // 调用 StreamTask 的构造方法
    super(env, null, FatalExitExceptionHandler.INSTANCE,
StreamTaskActionExecutor.synchronizedExecutor(lock));
    this.lock = Preconditions.checkNotNull(lock);

    // 这是 source 用于产生 data 的一个线程
    this.sourceThread = new LegacySourceFunctionThread();
}
```

我们首先来看 StreamTask 的具体构造方法的实现：

```
protected StreamTask(Environment environment, @Nullable TimerService
timerService, Thread.UncaughtExceptionHandler uncaughtExceptionHandler,
StreamTaskActionExecutor actionExecutor, TaskMailbox mailbox) throws Exception {

    .......

    // StrewamTask（ResultPartition + InputGate）
    // 创建 RecordWriter，大概率是：ChannelSelectorRecordWriter， 也有可能是个
BroadcastRecordWriter
    this.recordWriter = createRecordWriterDelegate(configuration, environment);

    // 初始化 StreamTask 的时候，初始化 MailboxProcessor， 同时，执行 StreamTask 的
processInput() 方法
    // 如果为 SourceStreamTask 的话，processInput 方法会启动 SourceStreamTask 的
sourceThread
```

```
    // Mail  MailBox  MailboxProcessor, mailboxProcessor 当中有一个方法叫做：
processMail(mail)
    this.mailboxProcessor = new MailboxProcessor(this::processInput, mailbox,
actionExecutor);

    // 创建 StateBackend，按照我们的配置，一般获取到的是 FsStateBackend
    this.stateBackend = createStateBackend();

    // 初始化 SubtaskCheckpointCoordinatorImpl 实例，主要作用是通过 StateBackend 创建
CheckpointStorage
    this.subtaskCheckpointCoordinator = new SubtaskCheckpointCoordinatorImpl(

        // 创建 CheckpointStorage，使用 FsStateBackend 的话，创建的就是
FsCheckpointStorage
        stateBackend.createCheckpointStorage(getEnvironment().getJobID()),
getName(), actionExecutor, getCancelables(),
        getAsyncOperationsThreadPool(), getEnvironment(), this,
configuration.isUnalignedCheckpointsEnabled(), this::prepareInputSnapshot);

    ......
}
```

其中在 SourceStreamTask 的 processInput() 方法中，主要是启动接收数据的线程 LegacySourceFunctionThread。

当构造方法完毕的时候，LegacySourceFunctionThread 已经初始化好了，但是 headOperator 还是 null，所以，LegacySourceFunctionThread 还未真正启动。

OneInputStreamTask 的构造器就没有什么特别的了，和 StreamTask 一样。

4.4. StreamTask 执行

4.4.1. SourceStreamTask 和 StreamTask 执行

接下来要进入到 StreamTask.invoke() 方法，核心分为四个步骤：

```
public final void invoke() throws Exception {

    // Task 正式工作之前
    beforeInvoke();

    // Task 开始工作：针对数据执行正儿八经的逻辑处理
    runMailboxLoop();

    // Task 要结束
    afterInvoke();

    // Task 最后执行清理
    cleanUpInvoke();
}
```

总结一下要点：

- 在 beforeInvoke() 中，主要是初始化 OperatorChain，然后调用 init() 执行初始化，然后恢复状态，更改 Task 自己的状态为 isRunning = true

- 在 runMailboxLoop() 中，主要是不停的处理 mail，这里是 FLink-1.10 的一项改进，使用了 mailbox 模型来处理任务
- 在 afterInvoke() 中，主要是完成 Task 要结束之前需要完成的一些细节，比如，把 Buffer 中还没 flush 的数据 flush 出来
- 在 cleanUpInvoke() 中，主要做一些资源的释放，执行各种关闭动作：set false，interrupt，shutdown，close，cleanup，dispose 等

当然，重点，一定是前两个步骤。

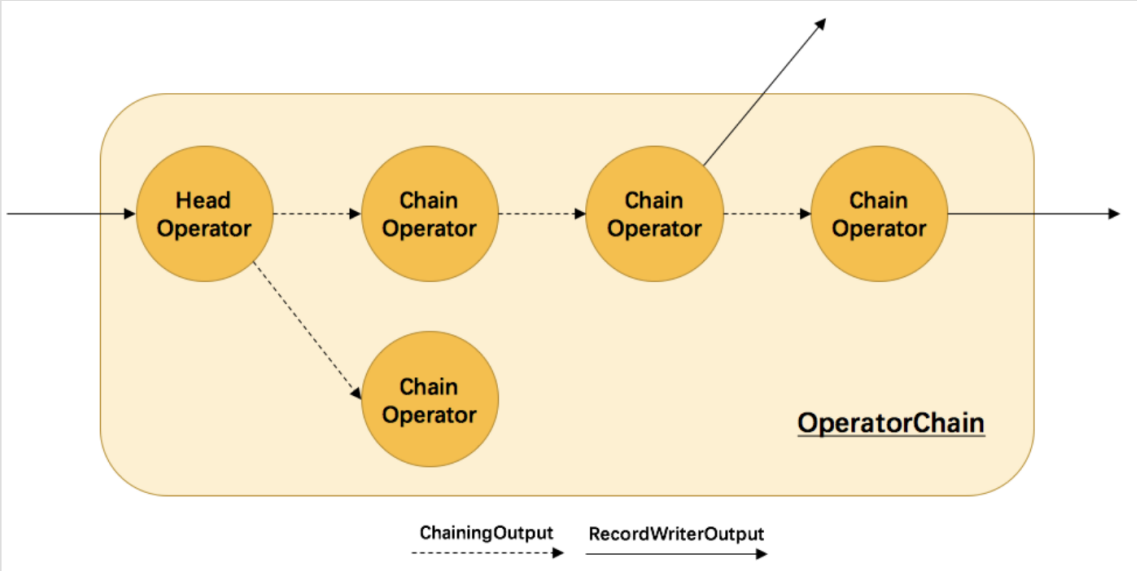
4.4.2. beforeInvoke() 执行细节

首先来看 beforeInvoke() 方法：

```
protected void beforeInvoke() throws Exception {

    // 初始化 OperatorChain
    operatorChain = new OperatorChain<>(this, recordWriter);

    // 获取 Head Operator
    headOperator = operatorChain.getHeadOperator();

    // 执行初始化， 原来讲解的是 Task 的初始化， 现在讲解的是 SourceStreamTask 的初始化
    SourceStreamTask.init();

    // 初始化状态
    actionExecutor.runThrowing(() -> {

 operatorChain.initializeStateAndOpenOperators(createStreamTaskStateInitializer(
));
         readRecoveredChannelState();
    });

    // 更改运行状态
    isRunning = true;
}
```

关于 状态恢复，我们留到最后一节知识再讲，先来看 构造 OperatorChain 和执行 SourceStreamTask 的初始化到底会做哪些工作！

首先看 ChainOperator 的初始化，首先会为每个 Operator 创建一个 RecordWriterOutput，再为每个 Operator 创建一个 OutputCollector。然后把每一个 Operator 都包装成 OperatorWrapper 放入 List<StreamOperatorWrapper> allOpWrappers 集合中。最后调用 linkOperatorWrappers(allOpWrappers); 方法以 逻辑正序 的方式来构建 StreamOperator 的链式关系。



然后是 init() 方法：

- 对于 SourceStreamTask 来说，就是看 Source 是不是 ExternallyInducedSource，如果是，则注册一个 savepoint 钩子。
- 对于 OneInputStreamTask 来说，主要就是创建 CheckpointedInputGate，StreamTaskNetworkOutput，StreamTaskNetworkInput，StreamOneInputProcessor 用来进行 Shuffle 相关的数据传输。

SourceStreamTask 的 init() 方法：

```
// 获取 SourceFunction
SourceFunction<?> source = headOperator.getUserFunction();

// 如果是外部诱导源，就注册一个 savepoint 钩子
if(source instanceof ExternallyInducedSource) {
    ExternallyInducedSource.CheckpointTrigger triggerHook = new
ExternallyInducedSource.CheckpointTrigger() {
        ....
    }
}
```

OneInputStreamTask 的 init() 方法：

```
// 创建 CheckpointedInputGate
CheckpointedInputGate inputGate = createCheckpointedInputGate();

// StreamTaskNetworkOutput 存在 StreamTask 中用于给下游 Task 输出结果数据的
DataOutput<IN> output = createDataOutput();

// StreamTaskNetworkInput 存在 StreamTask 中用于接收上游 Task 发过来的数据的
StreamTaskInput<IN> input = createTaskInput(inputGate, output);

// StreamOneInputProcessor 是 OneInputStreamTask 的 Input Reader
inputProcessor = new StreamOneInputProcessor<>(input, output, operatorChain);
```

到此为止，Task 初始化和预执行相关的，都基本到位了，然后就开始从我们的 SourceStreamTask 的 HeadOperator 的数据接收线程，开始流式处理。

4.4.3. runMailboxLoop() 执行细节

看源码调用栈：

```
runMailboxLoop();
    mailboxProcessor.runMailboxLoop();
        while(runMailboxStep(localMailbox, defaultActionContext)) {}
            // 处理 mail
            if(processMail(localMailbox)) {
                // 记录处理
                mailboxDefaultAction.runDefaultAction(defaultActionContext);
                    // 开始处理数据
                    StreamTask.processInput()
            }
```

注意此处的 StreamTask 有可能是 SourceStreamTask，有可能是 OneInputStreamTask。注意分别阅读！

SourceStreamTask 的 processInput() 方法内部细节：最重要的代码是

```
// 启动 SourceThread
sourceThread.start();
```

OneInputStreamTask 的 processInput() 方法内部细节：最重要的代码是

```
// 调用 OneInputStreamTask 的内部成员变量的 StreamInputProcessor 的 processInput() 执行处理
InputStatus status = inputProcessor.processInput();
    InputStatus status = StreamTaskInput.emitNext(output);
```

再继续看 LegacySourceFunctionThread sourceThread 的 start() 方法的实现，因为是个线程，具体逻辑在 run() 方法中。核心代码：

```
// SourceStreamTask 的 源数据处理线程开始工作
LegacySourceFunctionThread.run();
    // 调用 headOperator 的 run() 开始执行
    headOperator.run(lock, getStreamStatusMaintainer(), operatorChain);
        StreamSource.run();
            // 内部调用 UserFunction 的具体实现
            userFunction.run(ctx);
```

关于 UserFunction，则根据用户应用程序的具体实现，可以是不同的具体实现类！

在此，我们以 SocketTextStreamFunction 为例子，解读一下 run() 方法的具体实现：

```
// 初始化 BIO 的 Socket 客户端
Socket socket = new Socket();

// 获取读取数据的 输入流
BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

// 不停的读取数据，然后输出
while(isRunning && (bytesRead = reader.read(cbuf)) != -1) {
    // 输出数据
    ctx.collect(record);
}
```

输出数据之后的一些细节，请看源码解读！

4.4.4. afterInvoke() 执行细节

核心代码：主要是完成 Task 要结束之前需要完成的一些细节，比如，把 Buffer 中还没 flush 的数据 flush 出来

```
operatorChain.closeOperators(actionExecutor);
mailboxProcessor.prepareClose();
mailboxProcessor.drain();
operatorChain.flushOutputs();
disposeAllOperators(false);
disposedOperators = true;
```

4.4.5. cleanupInvoke() 执行细节

核心代码：主要做一些资源的释放，执行各种关闭动作：set false，interrupt，shutdown，close，cleanup，dispose 等

```
isRunning = false;
setShouldInterruptOnCancel(false);
Thread.interrupted();
tryShutdownTimerService();
cancelables.close();
shutdownAsyncThreads();
cleanup();
disposeAllOperators(true);
actionExecutor.run(() -> operatorChain.releaseOutputs());
channelIOExecutor.shutdown();
mailboxProcessor.close();
```

5. 本次内容总结

本次讲解的内容，是 Flink 的 StreamTask 的执行。

- StreamTask 分类： SourceStreamTask + OneInputStreamTask

- Task 的启动和执行逻辑，里面分为 10 个步骤，最重要的事情，就是初始化 Invokable 实例，并且后续调用 invoke() 执行启动
- SourceStreamTask 的初始化和执行
- OneInputStreamTask 的初始化和执行

如果你看 MR 的源码 MapTask， ReduceTask

如果你看 Spark 的源码： MapShuffleTask， ResultTask